ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

Empirical performance modeling of GPU kernels using active learning[1]

Prasanna Balaprakash[2], Karl Rupp[2], Azamat Mametjanov[2], Robert B. Gramacy[3], Paul D. Hovland[2], Stefan M. Wild[2]

Mathematics and Computer Science Division

Preprint ANL/MCS-P4097-0713

July 2013

[1]Support for this work was provided through the SciDAC program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract No. DE-AC02-06CH11357.

[2]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, USA

[3]Booth School of Business, University of Chicago

Empirical performance modeling of GPU kernels using active learning

Prasanna Balaprakash[1], Karl Rupp[1], Azamat Mametjanov[1]
Robert B. Gramacy[2], Paul D. Hovland[1], Stefan M. Wild[1]

Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439 [1]

Booth School of Business, University of Chicago [2]

Abstract

We focus on a design-of-experiments methodology for developing empirical performance models of GPU kernels. Recently, we developed an iterative active learning algorithm that adaptively selects parameter configurations in batches for concurrent evaluation on CPU architectures in order to build performance models over the parameter space. In this paper, we illustrate the adoption of the algorithm when concurrent evaluations are not possible, which is particularly useful in the absence of GPU clusters. We present an empirical study of the algorithm on a diverse set of GPU kernels and hardware. We show that even when concurrent evaluations are not possible, the default batch mode of the algorithm yields better models and the iterative active learning algorithm reduces the overall time required to obtain high-quality empirical performance models for GPU kernels.

1 Introduction

The introduction of general-purpose computations on graphics processing units (GPUs) has provided considerable performance gains for algorithms exposing fine-grained parallelism. The raw computing performance (in terms of floating-point operations per second) in concert with the memory bandwidth of recent hardware generations enable performance improvements of about an order of magnitude over conventional central processing units (CPUs). However, mapping even conceptually simple linear algebra operations, such as matrix-vector or matrix-matrix multiplications, to the underlying hardware can be difficult.

A key challenge is overcoming the high cardinality of the set of kernel instantiations, which vary, for example, in details such as block sizes for better cache reuse and the number of concurrent threads [9, 14]. Although empirical autotuning approaches have been employed for GPUs, these approaches can suffer from long execution times because of the high number of candidate compute kernels [7]. Consequently, performance portability of GPU computing can present a significant burden for application developers,

who seek to provide capabilities that will also work efficiently on a variety of architectures. Performance modeling is a promising approach for achieving this goal and may provide a much more efficient methodology for coping with high-dimensional search spaces.

We focus on a particular performance modeling task, where we predict the outputs of new configurations on an existing machine, a task that we distinguish from performance modeling tasks aimed at predicting performance on new architectures. When analytical performance models, which use mathematical abstractions for predicting performance metrics, become too restrictive, empirical performance modeling is an effective alternative. In empirical performance modeling, a set of parameter configurations (code variants) is evaluated on the target device to measure the required performance metrics, and a predictive model is built by using statistical approaches. We refer to these approximate models as surrogate models.

We previously developed an iterative algorithm, `ab-dynaTree`, that builds surrogate models for performance modeling using active learning [1]. The `ab-dynaTree` algorithm is based on dynamic regression trees [10], a methodology combining regression trees and Bayesian inference. Regression trees recursively partition a multidimensional input space into hyper-rectangles such that nearby inputs with similar output values fall within the same hyper-rectangle. Bayesian regression trees [3] are specified by a prior distribution on how the input space can be recursively partitioned, and a likelihood comprising a product of simple, independent regression models is applied in each partition. A dynamic regression [10] tree can perform inference for the tree sequentially, as input-output pairs become available. Dynamic trees are particularly useful in *active learning* contexts, namely, where heuristics are used to guide the selection of inputs at which outputs are gathered for model fitting. Active learning is an example of a *sequential design of experiments*.

The specific active learning heuristics used in this paper are presented in [1], and we ask the reader to refer to that paper for a detailed exposition of the resulting `ab-dynaTree` algorithm. A high-level description follows. At each iteration of `ab-dynaTree`, a dynamic tree model obtained through updates of the fits in previous iterations is used to choose the next inputs for training. The model-estimated average reduction in predictive variance is used to rank candidate configurations for inclusion in the training data. The novelty of `ab-dynaTree`, compared with similar methods described in [10], consists of choosing candidate parameter configurations in a sequence of multiconfiguration *batches*. Batching takes advantage of multinode cluster computing environments by evaluating multiple configurations concurrently,

thereby reducing the overall/wall-clock time required to obtain high-quality, fitted surrogate models. But choosing the right batch presents a number of challenges not addressed by earlier work.

In this paper, we apply `ab-dynaTree` to obtain surrogate models for GPU kernels when concurrent evaluations are not possible, that is, in the absence of availability of a GPU cluster. We present an empirical analysis of the `ab-dynaTree` algorithm with several GPU kernels and on different graphics cards and illustrate the algorithm's effectiveness in this context. The two main contributions of the paper are as follows:

- Empirical evidence that, because of the increased exploration capabilities, the default batch strategy in `ab-dynaTree` provides significant benefits over the classical sequential strategy even when concurrent evaluations are not possible
- Adoption of an active learning approach for obtaining surrogate models for GPU kernels and an empirical study spanning a diverse set of GPU kernels and graphic cards

The primary advantage of the iterative active learning algorithm is that it can significantly reduce the overall time required to obtain high-quality empirical performance models.

2 The GPU Kernels

Table 1 summarizes the GPU kernels and corresponding target hardware that we use to evaluate `ab-dynaTree`. We refer to *a problem* as a specific combination of a kernel and a GPU.

The problems `vc1`, `vc2`, `vc3`, and `vc4` correspond to an OpenCL implementation of the vector copy operation $x \leftarrow y$ (for vectors of size 4 million) with configurable local work group sizes and numbers of work groups at fine granularity. Both the increments of the local size and the number of work groups are considered in increments of 16 units. In addition, two different thread assignments are available. The first assigns a chunk of memory to all thread groups and then moves all work groups to the next chunk; in the second strategy, one work group operates on consecutive memory only.

The problem `vdot` performs the dot product of two vectors, $a \leftarrow x \cdot y$; `axpy` is a scaled vector addition, $y \leftarrow \alpha x + y$; and `spmv` is the product of a banded, sparse matrix and a vector, $y \leftarrow A_{dia} \times x$. Each of these kernels was implemented in CUDA. The performance-affecting parameters considered

Table 1: Problem set used for the experimental analysis.

Problems	Operations	Graphic Card	# Param.	Valid Configs. $\lvert\mathcal{X}_p\rvert$
vc1	vector copy	Nvidia GeForce GTX 285	4	2,560
vc2	vector copy	Nvidia GeForce GTX 470	4	2,560
vc3	vector copy	Nvidia Tesla C2050	4	2,560
vc4	vector copy	AMD Radeon HD 7970	4	2,560
vdot	vector dot product	Nvidia Tesla C2050	6	6,144
axpy	vector-scalar product	Nvidia Tesla C2050	6	7,680
spmv	sparse matrix-vector	Nvidia Tesla K20X	4	10,752
mm1	matrix multiplication	Nvidia GeForce GTX 470	10	8,465
mm2	matrix multiplication	AMD Radeon HD 7970	10	3,568

for each included the number of threads in a block, the number of blocks in a grid, the configurable size of the L1 cache, and compiler flags.

The problems mm1 and mm2 consist of dense matrix-matrix multiplication on two different GPUs. The OpenCL implementations of these kernels [11] reside in a ten-dimensional parameter space. These parameters allow for fine-grained control over block dimensions, loop unrolling, vector data types, and the use of on-chip shared memory. Although the total number of possible parameter configurations for each kernel is large (in the range of 20,000), many are "invalid" configurations, such as those exceeding the available shared memory or with block dimension mismatch. The invalidity of these configurations is known before empirical evaluation, and hence the invalid configurations were pruned from the parameter configuration space. The number of *valid* configurations $|\mathcal{X}_p|$ for each each problem is given in Table 1.

3 Experimental Results

We build surrogate models to minimize execution times of our benchmark problems. Since the kernels are executed in sequence on a single device and concurrent evaluation of parameter configurations is not possible, one can choose the classical sequential design-of-experiments approach that updates the model after each evaluation. This "serial version" of `ab-dynaTree` can be obtained by simply setting the batching parameter $n_b = 1$. However, the default mode of `ab-dynaTree` runs with $n_b > 1$; in the absence of concurrent evaluation capabilities, the model is updated only after these n_b serial evaluations have been performed.

In all experiments we run our `ab-dynaTree` algorithm with a maximum budget of $1,000$ evaluations. By $\mathcal{X}_{\text{out}}$ and $\mathcal{Y}_{\text{out}}$ we denote the set of 1,000

configurations and their corresponding execution times, respectively. To simplify our experimental setup and facilitate validation, for each problem we evaluate all valid configurations $\mathcal{X}_p$ and store the results $\mathcal{Y}_p$ in a lookup table, which is then queried by the algorithm and validation methodology.

To assess the quality of training points selected by `ab-dynaTree`, we use three regression algorithms implemented in `R` [4]: the `dynaTree` algorithm (dT) with 10 repetitions (as recommended by the package authors [5]) and taking the prediction at each x as the mean of the 10 predictions; random forest (rf) [2, 8], a state-of-the-art, robust, tree-based regression approach; and neural networks (nn) [13], a widely used nonlinear regression approach that has been used for empirical performance modeling in other settings [12]. For each algorithm, we consider two variants: *al*, in which points $(\mathcal{X}_{\text{out}}, \mathcal{Y}_{\text{out}})$ obtained from `ab-dynaTree` are used to train the three regression algorithms, and *rs*, in which 1,000 randomly selected points from $(\mathcal{X}_p,\ \mathcal{Y}_p)$ are used for train the regression algorithms. The default parameter values are used for dT and rf. Since the prediction accuracy of nn variants with default parameter values is poor, we run a computationally expensive subsidiary parameter tuning procedure [6] as suggested in [12].

We use the root-mean-squared error (RMSE) as a measure of prediction accuracy. To reduce the impact of randomness, we repeat each variant 10 times and consider the prediction accuracy of a variant as the RMSE averaged over 10 repetitions. A t-test is then applied to check whether the observed differences in the prediction accuracy of the variants are significant.

To allow cross-comparison of prediction accuracy between the problems, we scale the runtime values for each problem: each y_i is divided by $y_{i_{\max}}$, where $y_{i_{\max}}$ is the maximum execution time in $\mathcal{Y}_p$. For the active learning variants, we use 1,000 data points ($\mathcal{X}_{\text{out}}$,$\mathcal{Y}_{\text{out}}$) as the training set to build the surrogate model. We derive two test sets from the remaining points: (i) the subset of data points $\mathcal{T}_{25\%}$ whose mean run times are within the lower 25% quartile of the empirical distribution for the run times of $\mathcal{Y}_p$ and (ii) a set $\mathcal{T}_{100\%}$ of all remaining points. For the random sampling variants, we use the same test sets $\mathcal{T}_{25\%}$ and $\mathcal{T}_{100\%}$ but a different training set, with the 1,000 points for training being randomly chosen from $(\mathcal{X}_p, \mathcal{Y}_p) - \mathcal{T}_{25\%}$ (where the $\cdot - \cdot$ denotes a set difference).

First, we examine the impact of the batch size n_b in `ab-dynaTree` in terms of RMSE. To this end, we run tests with $n_b \in \{1, 50, 100, 200\}$. The training points obtained from `ab-dynaTree` are passed to dT(al), and each (different n_b) version starts with an initial sample of size 100. The results are shown in Figure 1. Using a large batch size results in `ab-dynaTree` allowing dT(al) to achieve lower RMSE with fewer training points. On 7 out of

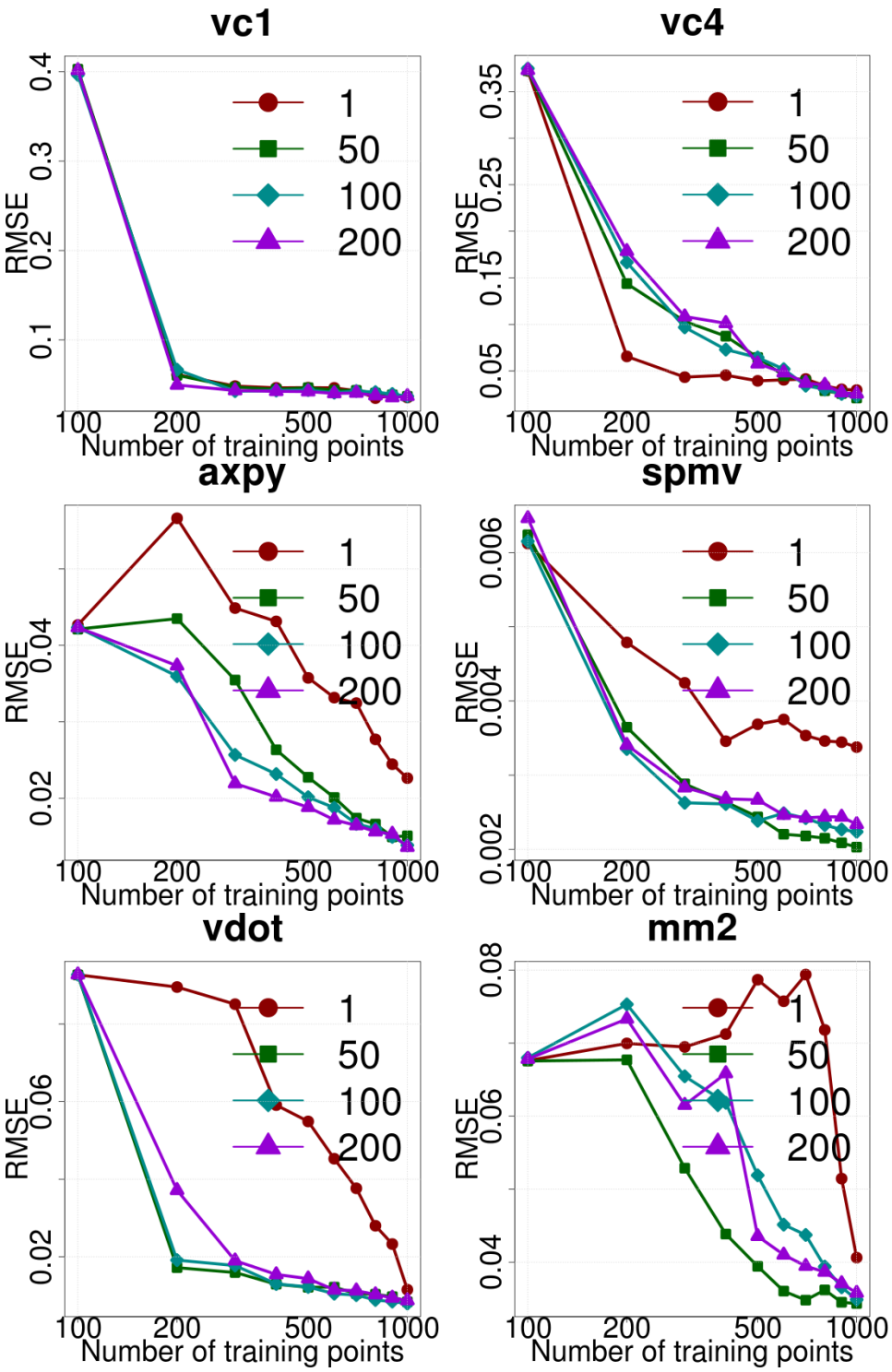

Figure 1: Average RMSE for different model update frequencies in `ab-dynaTree` on the test set $\mathcal{T}_{25\%}$.

Table 2: RMSE averaged over 10 replications on the $\mathcal{T}_{25\%}$ test set for 1,000 training points. The value is typeset in *italics* (**bold**) when a variant is significantly *worse* (**better**) than dT(al) according to a t-test with significance (alpha) level 0.05.

Problem	dT(al)	dT(rs)	nn(al)	nn(rs)	rf(al)	rf(rs)
vc1	0.035	*0.050*	0.036	*0.044*	*0.122*	*0.179*
vc2	0.041	*0.067*	0.039	*0.058*	*0.137*	*0.173*
vc3	0.124	*0.201*	0.131	*0.173*	*0.262*	*0.372*
vc4	0.026	*0.043*	*0.031*	*0.038*	*0.124*	*0.153*
vdot	0.009	*0.014*	0.010	*0.016*	*0.012*	*0.021*
axpy	0.014	*0.022*	0.012	*0.017*	*0.016*	*0.029*
spmv	0.002	*0.003*	0.002	*0.003*	*0.008*	*0.014*
mm1	0.029	*0.045*	0.027	*0.040*	0.028	*0.046*
mm2	0.036	*0.053*	**0.030**	*0.043*	**0.031**	*0.052*

the 9 problems, we find that updating the model immediately after each evaluation ($n_b = 1$) is not beneficial. This result can be attributed to the fact that large n_b values allow `ab-dynaTree` to explore and identify multiple regions in the input space with high-performing parameter configurations. However, with $n_b = 1$ `ab-dynaTree` has fewer exploration capabilities: there is a high probability of sampling from only one promising region of the input space. On `vc1`, the differences are negligible; but on `vc4`, `ab-dynaTree` with $n_b = 1$ outperforms those versions with larger batch sizes. In the rest of this section, we run `ab-dynaTree` with a batch size n_b of 200.

Now, we compare the three regression algorithms. Table 2 shows RMSEs averaged over 10 replications for 1,000 training points and tested on $\mathcal{T}_{25\%}$. We observe that the active learning (al) variants of dT, rf, and nn obtain lower RMSE than do their corresponding random sampling (rs) variants. The results also show that dT(al) completely dominates the three random sampling variants dT(rs), nn(rs), and rf(rs). Except for `mm2`, dT(al) obtains lower average RMSE than does rf(al). However, no statistically significant difference between dT(al) and nn(al) was detected by the t-test. The high performance of nn(al) can be attributed to the subsidiary parameter tuning procedure, which may be computationally infeasible in practical settings.

The key advantage of dT(al) is illustrated in Figure 2, which shows the RMSE as a function of the number of training points. We see that dT(al) requires relatively fewer evaluations to achieve a smaller RMSE. In Figure 3, we compare the number of evaluations required by the variants to reach the RMSE obtained by dT(rs) (with 1,000 evaluations). On 7 out of 9 problems, dT(al) reaches the RMSE of dT(rs) within 300 to 700 training points. Only

for mm1 and mm2 does rf(al) outperform dT(al).

Figure 4 shows the correlation between the observed and predicted values of dT(al) on $\mathcal{T}_{100\%}$. We note that dT(al) adopts an (optional) biased sampling procedure (see [1] for details) to model configurations with good performance and higher prediction accuracy by sacrificing the prediction accuracy of configurations with poor performance. Consequently, the accuracy of predicted values decreases with an increase in the observed values.

4 Conclusion

We have presented an experimental study of `ab-dynaTree`, an iterative active learning algorithm, for developing empirical performance models of GPU kernels. We demonstrated that even when concurrent evaluations of code variants are not possible, the default batch mode of `ab-dynaTree`, which allows for a higher degree of parameter space exploration, provides significant benefits over the classical, serial mode of `ab-dynaTree`. We showed that the `ab-dynaTree` algorithm can significantly reduce the overall time (up to a factor of 3 when compared with naïve random sampling) required to obtain empirical performance models for GPU kernels.

Our future work for `ab-dynaTree` includes asynchronous model updates, multiobjective surrogate modeling (e.g., for run time, power consumption, bandwidth, and FLOPS simultaneously) with a single run of `ab-dynaTree`, and deployment of `ab-dynaTree` within autotuning search algorithms.

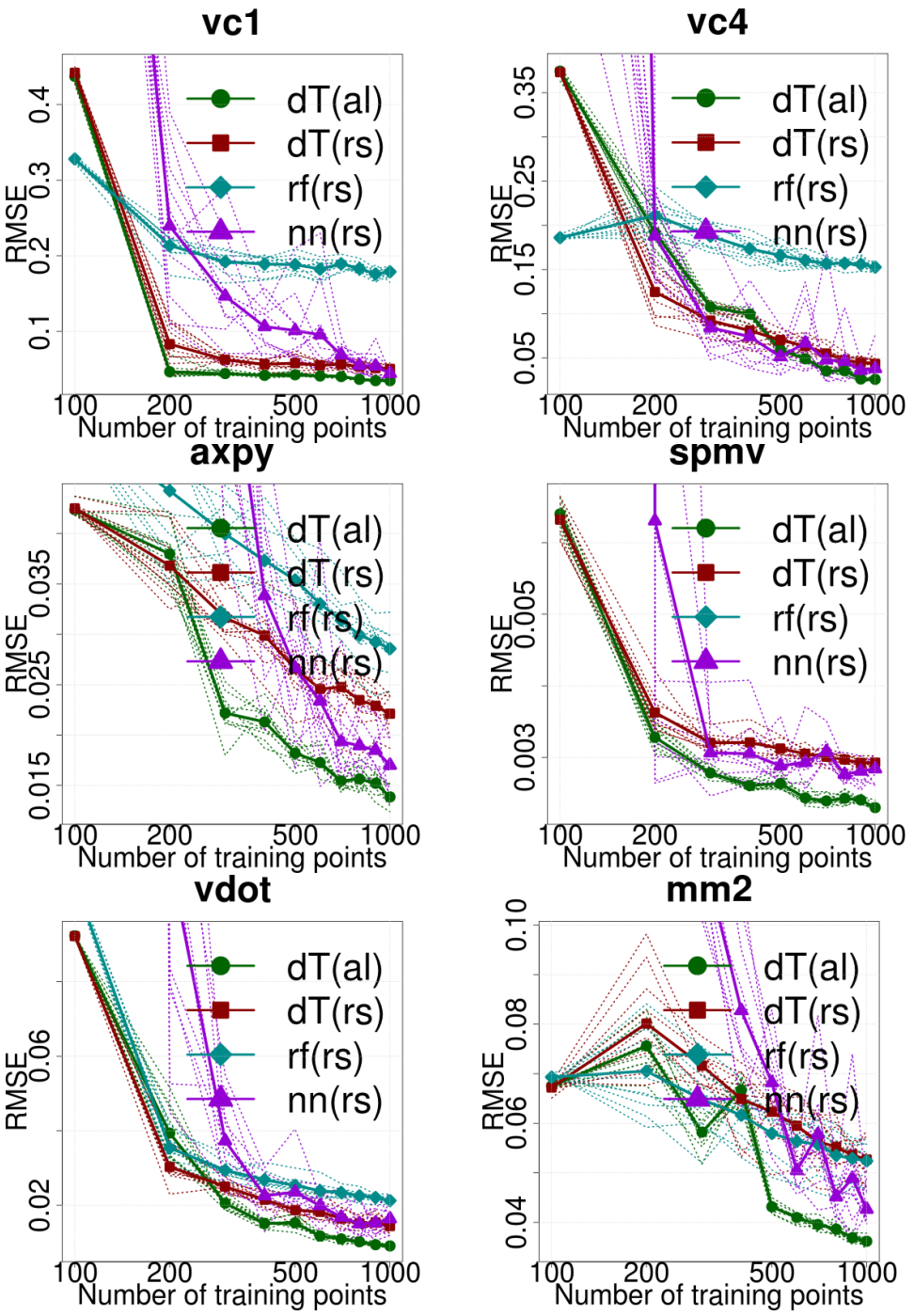

Figure 2: RMSE for various GPU kernels on the test set $\mathcal{T}_{25\%}$. The dotted lines represent the RMSE for each replication, and the bold lines represent the mean RMSE over 10 replications.9

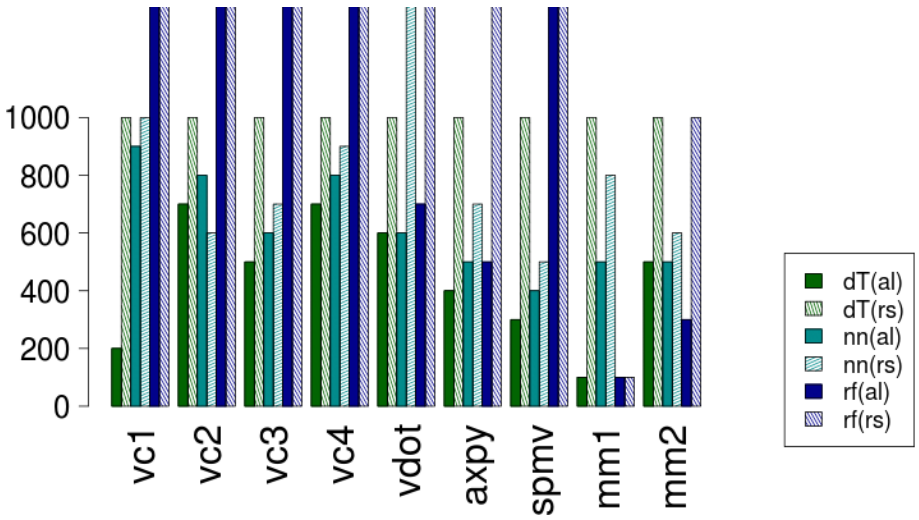

Figure 3: Number of evaluations required to reach the RMSE of dT(rs) (with 1,000 evaluations) on the test set $\mathcal{T}_{25\%}$.

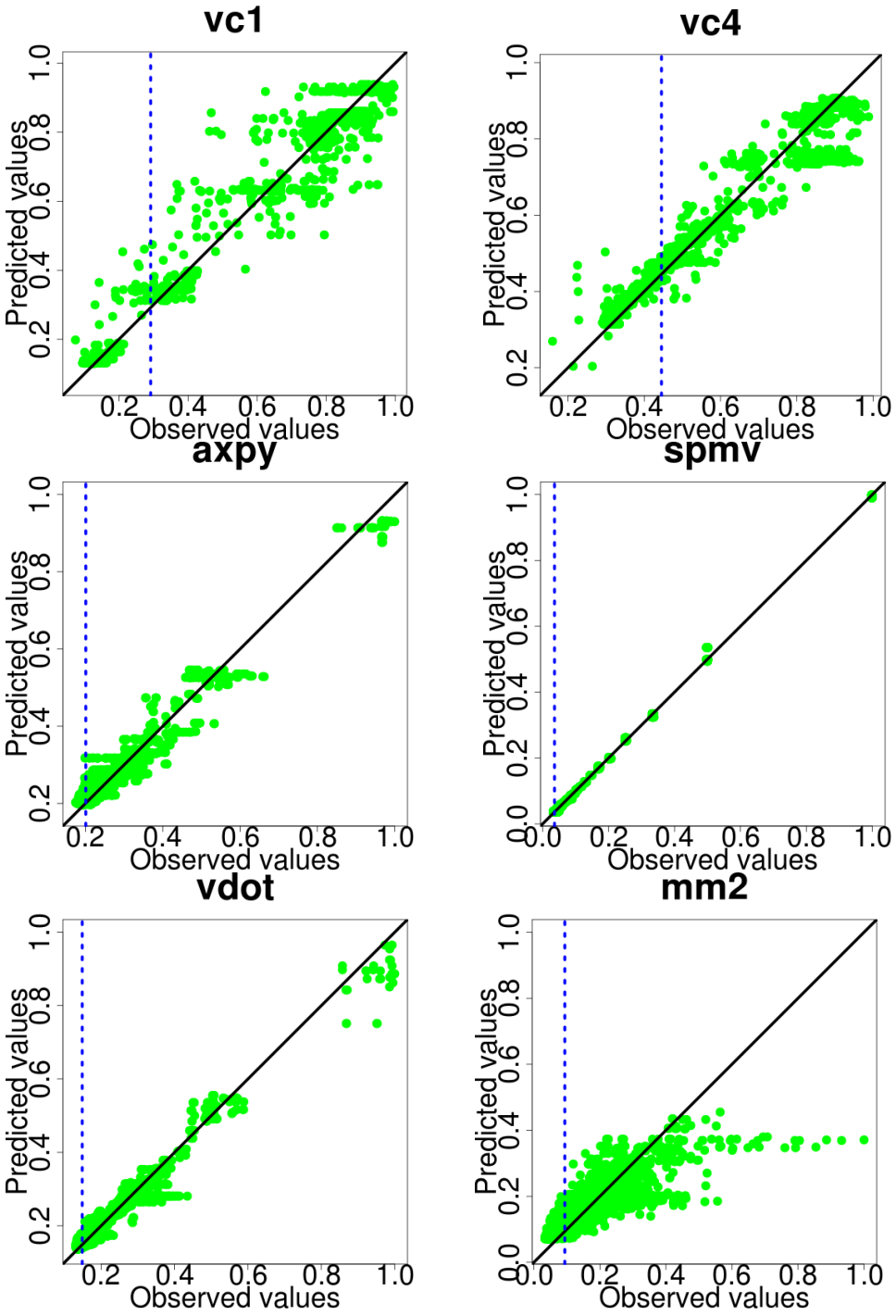

Figure 4: Correlation plot for GPU kernels on $\mathcal{T}_{100\%}$. The vertical, dotted line represents the 25% quantile. The number of testing points $|\mathcal{T}_{100\%}|$ in each plot is equal to $|\mathcal{X}_p| - 1000$.

Acknowledgment

This work was supported by the U. S. Department of Energy, Office of Science, under Contract No. DE-AC02-06CH11357.

References

[1] P. Balaprakash, R. B. Gramacy, and S. M. Wild. Active-learning-based surrogate models for empirical performance tuning. In IEEE International Conference on Cluster Computing (CLUSTER), 2013.

[2] L. Breiman. Random forests. Machine Learning, 45(1):5–32, 2001.

[3] H. Chipman, E. George, and R. McCulloch. Bayesian treed models. Machine Learning, 48:303–324, 2002.

[4] R. Gentleman, R. Ihaka, D. Bates, et al. The R project for statistical computing, 1997. R website: http://www. r-project. org.

[5] R. B. Gramacy and M. A. Taddy. `dynaTree`: Dynamic Trees for Learning and Design, 2011. R package version 2.0.

[6] M. Kuhn. Building predictive models in R using the caret package. Journal of Statistical Software, 28(5):1–26, 2008.

[7] J. Kurzak, S. Tomov, and J. Dongarra. Autotuning GEMM kernels for the Fermi GPU. IEEE Transactions on Parallel and Distributed Systems, 23(11):2045–2057, 2012.

[8] A. Liaw and M. Wiener. Classification and regression by randomForest. R news, 2(3):18–22, 2002.

[9] K. Matsumoto, N. Nakasato, and S. G. Sedukhin. Implementing a code generator for fast matrix multiplication in OpenCL on the GPU. In 6th IEEE International Symposium on Embedded Multicore SoCs, pages 198–204, 2012.

[10] M. A. Taddy, R. B. Gramacy, and N. G. Polson. Dynamic trees for learning and design. Journal of the American Statistical Association, 106(493):109–123, 2011.

[11] P. Tillet, K. Rupp, S. Selberherr, and C.-T. Lin. Towards performance-portable, scalable, and convenient linear algebra. In USENIX Workshop on Hot Topics in Parallelism (HotPar'13), pages 1–8, 2013.

[12] A. Tiwari, M. A. Laurenzano, L. Carrington, and A. Snavely. Modeling power and energy usage of HPC kernels. In IEEE International Symposium on Parallel & Distributed Processing Workshops and PhD Forum (IPDPSW12), pages 990–998, 2012.

[13] W. N. Venables and B. D. Ripley. Modern Applied Statistics with S-PLUS, volume 250. Springer-verlag New York, 1994.

[14] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, pages 31:1–31:11, 2008.